下载APP

04 | 封闭类：怎么刹住失控的扩展性？

2021-11-22 范学雷

《深入剖析Java新特性》 课程介绍 >

讲述：范学雷

时长 14:55 大小 13.67M

你好，我是范学雷。今天，我们聊一聊 Java 的封闭类。

封闭类这个特性，首先在 JDK 15 中以预览版的形式发布。在 JDK 16 中，改进的封闭类再次以预览版的形式发布。最后，封闭类在 JDK 17 正式发布。

那么，什么是封闭类呢？封闭类的英文，使用的词汇是"sealed classes"。从名字我们就可以感受到，封闭类首先是 Java 的类，然后它还是封闭的。

Java 的类，我们都知道什么意思。那么，“封闭”又是什么意思呢？字面的意思，就是把一些东西封存起来，里面的东西出不去，外面的东西也进不来，所以可查可数。

“封闭”、“可查可数”，这些词汇字面看起来好像很通俗，但是实际上并不容易理解。我们还是通过案例和代码，一步一步地来了解封闭类吧。

阅读案例

在面向对象的编程语言中，研究表示形状的类，是一个常用的教学案例。今天的评审案例，我们也从形状这个类开始，来研究一下怎么判断一个形状是不是正方形吧。

下面的这段代码，就是一个简单的、抽象的形状类的定义。这个抽象类的名字是 **Shape**。它有一个抽象方法 area()，用来计算形状的面积。它还有一个公开的属性 id，用来标识这个形状的对象。

复制代码

```
package co.ivi.jus.sealed.former;

public abstract class Shape {
    public final String id;

    public Shape(String id) {
        this.id = id;
    }

    public abstract double area();
}
```

我们都知道，正方形是一个形状。正方形可以作为形状这个类的一个扩展类。它的代码可以是下面的样子。

复制代码

```
package co.ivi.jus.sealed.former;

public class Square extends Shape {
    public final double side;

    public Square(String id, double side) {
        super(id);
        this.side = side;
    }

    @Override
    public double area() {
        return side * side;
```

```
    }
}
```

那么，到底怎么判断一个形状是不是正方形呢？这个问题的答案，表面上看起来很简单，只要判断这个形状的对象是不是一个正方形的实例就可以了。这个判断的例子，看起来可以是下面的样子。

复制代码

```
static boolean isSquare(Shape shape) {
    return (shape instanceof Square);
}
```

你可以思考一下，这样是不是真的能判断一个形状是正方形？花几秒钟想想你的答案，我们接下来再继续分析。

案例分析

其实，上面的这个例子，判断的只是“一个形状的对象是不是一个正方形的实例”。但实际上，一个形状的对象即使不是一个正方形的类，它也有可能是一个正方形。什么意思呢？比如说有一个对象，表示它的类是长方形或者菱形的类。如果这个对象的每一个边的长度都是一样的，其实它就是一个正方形，但是表示它的类是长方形或者菱形的类，而不是正方形类。所以，上面的这段代码还是有缺陷的，并不总是能够正确判断一个形状是不是正方形。

详细地，我们来看下一段代码，你就对这个缺陷有一个更直观的了解了。我们都知道，长方形也是一个形状，它也可以作为形状这个类的一个扩展类。下面的这段代码，定义的就是一个长方形。这个类的名字是 **Rectangle**，它是 **Shape** 的扩展类。

复制代码

```
package co.ivi.jus.sealed.former;

public class Rectangle extends Shape {
    public final double length;
    public final double width;

    public Rectangle(String id, double length, double width) {
        super(id);
        this.length = length;
```

```
        this.width = width;
    }

    @Override
    public double area() {
        return length * width;
    }
}
```

代码读到这里，对于“怎么判断一个形状是不是正方形”这个问题，我觉得你可能已经有了一个更好的思路。没错，正方形是一个特殊的长方形。如果一个长方形的长和宽是相等的，那么它也是一个正方形。上面的那段“判断一个形状是不是正方形”的代码，就没有考虑到长方形的特例，所以它是有缺陷的实现。

知道了长方形这个类，我们就能改进我们的判断了。改进的代码，要把长方形考虑进去。它看起来可以是下面的样子。

复制代码

```
public static boolean isSquare(Shape shape) {
    if (shape instanceof Rectangle rect) {
        return (rect.length == rect.width);
    }

    return (shape instanceof Square);
}
```

写完上面的代码，似乎就可以长舒一口气：哎，这难缠的正方形，我们终于搞定了。

但其实，这个问题我们还没有搞定。因为正方形也是一个特殊的菱形，如果一个对象是一个菱形类的实例，上面的代码就有缺陷。更令人窘迫的是，正方形还是一个特殊的梯形，还是一个特殊的多边形。随着我们学习一步一步的深入，我们知道还有很多形状的特殊形式是正方形，而且我们并不知道我们知识范围外的那些形状，当然更不能提穷举它们了。

这，实在有点让人抓狂！

问题出在哪里呢？**无限制的扩展性，是问题的根源**。正如现实世界里，我们没有办法穷举到底有多少形状的特殊形式是正方形；在计算机的世界里，我们也没有办法穷举到底有多

少形状的对象可以是正方形。如果我们解决不了形状类的穷举问题，我们就不太容易使用代码来判断一个形状是不是正方形。

而解决问题的办法，就是限制可扩展类的扩展性。

怎么限制住扩展性？

你可能要问，可扩展性不是面向对象编程的一个重要指标吗？为什么要限制可扩展性呢？其实，面向对象编程的最佳实践之一，就是要把可扩展性限制在可以预测和控制的范围内，而不是无限的可扩展性。

除了上面穷举的问题之外，在极客时间专栏《代码精进之路》里，我们还讨论了继承的安全缺陷。其中，主要有两点值得我们格外小心：

> 一个可扩展的类，子类和父类可能会相互影响，从而导致不可预知的行为。
> 涉及敏感信息的类，增加可扩展性不一定是个优先选项，要尽量避免父类或者子类的影响。

虽然我们使用了 Java 语言来讨论继承的问题，但其实这些是面向对象机制的普遍问题，甚至它们也不单单是面向对象语言的问题，比如使用 C 语言的设计和实现，也存在类似的问题。

由于继承的安全问题，我们在设计 API 时，有两个要反省思考的点：

> 一个类，有没有真实的可扩展需求，能不能使用 final 修饰符？
> 一个方法，子类有没有重写的必要性，能不能使用 final 修饰符？

限制住不可预测的可扩展性，是实现安全代码、健壮代码的一个重要目标。

JDK 17 之前的 Java 语言，限制住可扩展性只有两个方法，使用私有类或者 final 修饰符。显而易见，私有类不是公开接口，只能内部使用；而 final 修饰符彻底放弃了可扩展性。要么全开放，要么全封闭，可扩展性只能在可能性的两个极端游走。全封闭彻底没有了可扩展性，全开放又面临固有的安全缺陷，这种二选一的状况有时候很让人抓狂，特别是设计公开接口的时候。

JDK 17 之后，有了第三种方法。这个办法，就是使用 Java 的 **sealed** 关键字。使用类修饰符 **sealed** 修饰的类是封闭类；使用类修饰符 **sealed** 修饰的接口是封闭接口。封闭类和封闭接口限制可以扩展或实现它们的其他类或接口。

通过把可扩展性的限制放在可以预测和控制的范围内，封闭类和封闭接口打开了全开放和全封闭两个极端之间的中间地带，为接口设计和实现提供了新的可能性。

怎么声明封闭类

那么，怎么使用封闭类呢？封闭类这个概念，涉及到两种类型的类。第一种是被扩展的父类，第二种是扩展而来的子类。通常地，我们把第一种称为封闭类，第二种称为许可类。

封闭类的声明使用 sealed 类修饰符，然后在所有的 extends 和 implements 语句之后，使用 permits 指定允许扩展该封闭类的子类。 比如，使用 sealed 类修饰符，我们可以把形状这个类声明为封闭类。下面的这个例子中，Shape 是一个封闭类，可以扩展它的子类只有两个，分别为 Circle 和 Square。也就是说，这里定义的形状这个类，只允许有圆形和正方形两个子类。

复制代码

```
package co.ivi.jus.sealed.modern;

public abstract sealed class Shape permits Circle, Square {
    public final String id;

    public Shape(String id) {
        this.id = id;
    }

    public abstract double area();
}
```

由 permits 关键字指定的许可子类（permitted subclasses），必须和封闭类处于同一模块（module）或者包空间（package）里。如果封闭类和许可类是在同一个模块里，那么它们可以处于不同的包空间里，就像下面的例子。

复制代码

```
package co.ivi.jus.sealed.modern;

public abstract sealed class Shape
```

```
    permits co.ivi.jus.ploar.Circle,
            co.ivi.jus.quad.Square {
    public final String id;

    public Shape(String id) {
        this.id = id;
    }

    public abstract double area();
}
```

如果允许扩展的子类和封闭类在同一个源代码文件里，封闭类可以不使用 permits 语句，Java 编译器将检索源文件，在编译期为封闭类添加上许可的子类。比如下面的两种 Shape 封闭类的声明，一个封闭类使用了 permits 语句，另外一个封闭类没有使用 permits 语句。但是，这两个声明具有完全一样的运行时效果。

复制代码

```
package co.ivi.jus.sealed.improved;

public abstract sealed class Shape {
    public final String id;

    public Shape(String id) {
        this.id = id;
    }

    public abstract double area();

    public static final class Circle extends Shape {
        // snipped
    }

    public static final class Square extends Shape {
        // snipped
    }
}
```

复制代码

```
package co.ivi.jus.sealed.improved;

public abstract sealed class Shape
         permits Shape.Circle, Shape.Square {
    public final String id;

    public Shape(String id) {
```

```
        this.id = id;
    }

    public abstract double area();

    public static final class Circle extends Shape {
        // snipped
    }

    public static final class Square extends Shape {
        // snipped
    }
}
```

不过，如果你读过《代码精进之路》，你就会倾向于总是使用 permits 语句。因为这样的话，代码的阅读者不需要去翻找上下文，也能一目了然地知道这个封闭类支持哪些许可类。这会给代码的阅读者带来很多的便利，包括节省时间以及少犯错误。

怎么声明许可类

许可类的声明需要满足下面的三个条件：

> 许可类必须和封闭类处于同一模块（module）或者包空间（package）里，也就是说，在编译的时候，封闭类必须可以访问它的许可类；
>
> 许可类必须是封闭类的直接扩展类；
>
> 许可类必须声明是否继续保持封闭：
>
> - 许可类可以声明为终极类（final），从而关闭扩展性；
> - 许可类可以声明为封闭类（sealed），从而延续受限制的扩展性；
> - 许可类可以声明为解封类（non-sealed），从而支持不受限制的扩展性。

比如在下面的例子中，许可类 Circle 是一个解封类；许可类 Square 是一个封闭类；许可类 ColoredSquare 是一个终极类；而 ColoredCircle 既不是封闭类，也不是许可类。

复制代码

```
package co.ivi.jus.sealed.propagate;

public abstract sealed class Shape {
    public final String id;
```

```
    public Shape(String id) {
        this.id = id;
    }

    public abstract double area();

    public static non-sealed class Circle extends Shape {
        // snipped
    }

    public static sealed class Square extends Shape {
        // snipped
    }

    public static final class ColoredSquare extends Square {
        // snipped
    }

    public static class ColoredCircle extends Circle {
        // snipped
    }
}

```

需要注意的是，**由于许可类必须是封闭类的直接扩展，因此许可类不具备传递性。**也就是说，上面的例子中，ColoredSquare 是 Square 的许可类，但不是 Shape 的许可类。

案例回顾

到这里，我们再回头看看前面的案例，怎么判断一个形状是不是正方形呢？封闭类能帮助我们解决这个问题吗？如果使用了封闭类，这个问题的答案也就呼之欲出了。

首先，我们要把形状这个类定义为封闭类。这样，所有形状的子类就可以穷举了。然后，我们寻找可以用来表示正方形的许可类。找到这些许可类后，只要我们能够判断这个形状的对象是不是一个正方形，问题就解决了。

比如下面的代码，形状被定义为封闭类 Shape。而且，Shape 这个封闭类只有两个终极的许可类。一个许可类是表示圆形的 Circle，一个许可类是表示正方形的 Square。

复制代码

```
package co.ivi.jus.sealed.improved;

public abstract sealed class Shape
```

```
4           permits Shape.Circle, Shape.Square {
5     public final String id;
6
7     public Shape(String id) {
8         this.id = id;
9     }
10
11    public abstract double area();
12
13    public static final class Circle extends Shape {
14        // snipped
15    }
16
17    public static final class Square extends Shape {
18        // snipped
19    }
20 }
```

由于 Shape 是个封闭类，在这段代码的许可范围内，一个形状 Shape 的对象要么是一个圆形 Circle 的实例，要么是一个正方形 Square 的实例，没有其他的可能性。

这样的话，判断一个形状是不是正方形这个问题就变得比较简单了。只要能够判断出来一个形状的对象是不是一个正方形的实例，这个问题就算是解决了。

复制代码

```
1 static boolean isSquare(Shape shape) {
2     return (shape instanceof Square);
3 }
```

这样的逻辑在案例分析那一小节的场景中并不成立，为什么现在就成立了呢？根本的原因，在案例分析那一小节的场景中，Shape 类是一个不受限制的类，我们没有办法知道它所有的扩展类，因此我们也就没有办法穷尽正方形的所有可能性。而在使用封闭类的场景下，Shape 类的所有扩展类，我们都是已知的，所以我们就有办法检查每一个扩展类的规范，从而对这个问题做出正确的判断。

总结

好，到这里，我来做个小结。从前面的讨论中，我们了解到，可扩展性的限定方法有四个：

1. 使用私有类；
2. 使用 final 修饰符；
3. 使用 sealed 修饰符；
4. 不受限制的扩展性。

在我们日常的接口设计和编码实践中，使用这四个限定方法的优先级应该是由高到低的。最优先使用私有类，尽量不要使用不受限制的扩展性。

如果要丰富你的代码评审清单，有了封闭类后，你可以加入下面这一条：

> 一个类，如果有真实的可扩展需求，能不能枚举，可不可以使用 sealed 修饰符？

另外，通过今天的讨论，我拎出几个技术要点，这些都可能在你们面试中出现哦，通过学习，你应该能够：

知道 Java 支持封闭类，并且能够使用封闭类编写代码；

- 面试问题：你知道封闭类吗？会不会使用它？

了解封闭类的原理和它要解决的问题，知道限制住扩展性的办法；

- 面试问题：面向对象编程的可扩展性有什么问题吗？该怎么处理这些问题？

能够有意识地使用封闭类来限制类或者接口的扩展性。

- 面试问题：你写的这段代码，是不是应该使用 **final** 修饰符或者 **sealed** 修饰符？

如果你的代码里使用了封闭类，无论是面试的时候还是工作的时候，一定能够给人深刻的印象。因为，这意味着你已经了解了可扩展性的危害，并且有办法降低这种危害的影响，有能力编写出更健壮的代码。

思考题

在案例回顾这一小节里，我们使用了封闭类来解决“怎么判断一个形状是不是正方形”这个问题。我们假设案例回顾这一小节的代码是版本 1.0。现在我们假设，在版本 2.0 里，需要增加另一个许可类，用来支持长方形（Rectangle）。那么：

1. 封闭类的代码该怎么改动，才能支持长方形？
2. “判断一个形状是不是正方形”的代码该怎么改动，才能适应封闭类的改变？
3. 增加一个许可类，会有兼容性的影响吗？比如说，使用版本 1.0 来判断一个形状是不是正方形的代码还能使用吗？

欢迎你在留言区留言、讨论，分享你的阅读体验以及对这些问题的思考。

注：本文使用的完整的代码可以从GitHub下载，你可以通过修改GitHub上review template代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在封闭类专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 sealed/review/xuelei 的目录下面。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 03 | 档案类：怎么精简地表达不可变数据？

下一篇 05 | 类型匹配：怎么切除臃肿的强制转换？

精选留言 (10)

写留言

许灵

2021-11-22

https://github.com/XueleiFan/java-up/pull/7

展开

作者回复: 代码干净利索！

2

雷霹雳的爸爸
2021-11-22

1. permits 得加上Rectangle，2. 要增加instance of Rectangle时候，长和宽是不是相等的逻辑，3. 正方形那段还是生效的，但是如果shape是Rect就不起作用了，所以还是应该认为1.0得isSquare不兼容新增加的许可类

假设我3判断的没错（确实还没去code...），那么我的问题也在这，因为岂不是这个例子...
展开

作者回复: 1）2）3）的判断都是对的。第3个问题，其实是我们讨论问题的重点。增加一个抽象方法当然是好办法，不过也有很多例子没有办法通过增加父类的方法来解决。封闭类需要和模式匹配一起使用，才能解决好问题3。模式匹配带来的变化，我们后面会讨论。

共 2 条评论 1

Geek_956c95
2021-11-30

需要注意的是，由于许可类必须是封闭类的直接扩展，因此许可类不具备传递性。也就是说，上面的例子中，ColoredSquare 是 Square 的许可类，但不是 Shape 的许可类。
是不是 Square应该写成 Circle
也就是说，上面的例子中，ColoredSquare 是 Circle 的许可类，但不是 Shape 的许可类。

展开

作者回复: 该怎么理解ColoredSquare和Circle的关系（ColoredSquare extends Square）？

ABC
2021-11-23

我在openjdk的githubca仓库提交人员列表中看到老师了,https://github.com/openjdk/jdk/graphs/contributors,太强了,给老师点赞!

展开

作者回复: 哈哈，只是一部分工作。

fatme
2021-11-23

如果一开始设计封闭类的时候，对许可子类的预估不足，造成后续要增加新的许可子类。这样就要修改 permits 代码，这不太方便，也违背开闭原则。permits 是否能支持通配符呢？这样同一个包的所有直接子类都是许可类，在一定程度上可以减少需要修改 permits 的情况。

展开

作者回复: 这是一个权衡和妥协，封闭类的好处，可能要到第10讲，我们才能理解清楚。

ABC
2021-11-22

1. permits需要增加新增的类;2和3都用实例模式匹配修改了一下.源码里面删掉了isSquare方法,考虑向下兼容可以在isSquare方法中调用新增的getInstanceName方法进行兼容.

用了返回实例类名的方式来返回指定类,以此判断,如果还有更好的办法,请老师指点.

...

展开

作者回复: 代码整体很好，只是我还有一个小的疑问，在PR里留言了。

共 2 条评论

Jxin
2021-11-22

课后题
3.向前兼容，向后不兼容。向前因为没有长方形的场景，所以是兼容的；向后因为有了长方形的场景，但没有对应的 是否正方形的判断逻辑，所以不兼容。但1.0的实现本身就不合适，这个是否正方形的实现应该要跟着子类走，也就是以抽象方法要求子类必须做实现。...

展开

作者回复: 相对于继承，我更害怕组合的缺陷。

现实可能没有这么理想。 实际的情况往往是我们设计了1.0，用户实现了它们的逻辑，然后我们想

在2.0里增加些什么（比如一个新方法），对用户来说已经太晚了（除非他知道并且愿意改代码）。

共 2 条评论 >

黄剑豪

2021-11-22

感觉这个例子举得不好，会让人不太理解使用场景，因为通常来说Shape本身不应该是个封闭场景。因为判断是否是正方形强行将其封闭，总觉得例子有点怪。

作者回复: 嗯。 有没有什么好的例子的建议？选例子，可能是我花时间最多的地方，也是比较头疼的。

共 2 条评论 >

密码123456

2021-11-22

我是在，现场进行开发的。一般都是，公司内出版本。客户现场接增量需求。但是没有，公司内部版本的源码。如果有这么封闭类，有些定制化需求，客户要的急，需要改版本的源码，这样是不是相当于把扩展性的通道，堵死了？

展开

作者回复: 技术当然要服从于现实的需求。如果你的场景需要无限的扩展性，那当然就只能使用无限的扩展性。封闭类有它适合的场景，也有不便使用的地方。需求的扩展和类的扩展，有重合的部分，但是也可能要具体情况具体分析。

aoe

2021-11-22

原来正方形变化多端！封闭类可以把无限多种情况变成有限数量的情况，从而达到了不可控到可控的目的。虽然在添加新许可类时必须修改封闭类（打破了开闭原则），但解决了因无限扩展可能导致的诡异Bug，保证的程序的正确性（没有比正确运行更重要的了）。

展开

作者回复: "没有比正确运行更重要的了"! 这是一切原则能够成立的根本啊。